In the Specification:

The application stands objected to because a computer program listing of over sixty (60) lines and less than three hundred-one (301) lines within the written specification must be positioned at the end of the specification and before the claims. The following amendments address this objection:

Please insert the substitute sheet, included with this response and containing the computer listing of paragraph **[0028]**, at the end of the detailed description but before the claims.

In the specification, please replace paragraph **[0028]** with the following paragraph:

[0028] The ~~following are the~~ function stubs (see step 160) for registering the Linux device driver structure [[:]] are shown in Computer Listing A at the end of this detailed description.

~~STATIC loff_t sym_lseek(struct file *f, loff_t off, int a)~~
~~{ m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__);~~
~~return(ENODEV); }~~
~~STATIC ssize_t sym_read(struct file *f, char *c, size_t b, loff_t * a)~~
~~{ m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__);~~
~~return(ENODEV); }~~
~~STATIC ssize_t sym_write(struct file *f, const char *c,size_t b,loff_t * a)~~
~~{ m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__);~~
~~return(ENODEV); }~~
~~STATIC int sym_readdir(struct file *f, void *v, filldir_t dir)~~
~~{ m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__);~~
~~return(ENODEV); }~~
~~STATIC unsigned int sym_poll(struct file *f, poll_table *poll)~~
~~{ m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__);~~
~~return(ENODEV); }~~
~~STATIC int sym_ioctl(struct inode *i, struct file *f, unsigned int cmd,~~
~~unsigned long arg)~~
~~{ m_printk("%s: unsupported function %s cmd %d \n",MODULE_NAME,~~
~~__FUNCTION__, cmd);~~
~~return(ENODEV); }~~
~~STATIC int sym_mmap(struct file *f, struct vm_area_struct *vm)~~
~~{ m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__);~~

```
        return(ENODEV); }
STATIC int sym_open ( struct inode *i, struct file *f)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__ );
        return(ENODEV); }
STATIC int sym_flush( struct file *f)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__ );
        return(ENODEV); }
STATIC int sym_release (struct inode *i, struct file *f)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__ );
        return(ENODEV); }
STATIC int sym_fsync( struct file *f, struct dentry *d)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__ );
        return(ENODEV); }
STATIC int sym_fasync(int b, struct file *f, int a)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__ );
        return(ENODEV); }
STATIC int sym_check_media_change( kdev_t dev )
{       m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__ );
        return(ENODEV); }
STATIC int sym_revalidate( kdev_t dev )
{       m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__ );
        return(ENODEV); }
STATIC int sym_lock( struct file *f, int a, struct file_lock *l )
{       m_printk("%s: unsupported function %s \n",MODULE_NAME, __FUNCTION__ );
        return(ENODEV); }

STATIC struct file_operations sym_opts =
{
        sym_lseek,
        sym_read,
        sym_write,
        sym_readdir,
        sym_poll,
        sym_ioctl,
        sym_mmap,
        sym_open,
        sym_flush,
        sym_release,
        sym_fsync,
        sym_fasync,
        sym_check_media_change,
        sym_revalidate,
        sym_lock
};
```

Furthermore, please amend the specification as follows:

Please replace paragraph **[0002]** with the following amended paragraph:

[0002] In computer systems, a device driver is typically used to interface various different software applications to a particular hardware device or peripheral. The driver thus provides an interface to a hardware device so that it can perform functions requested by a variety of different application packages. For example, the applications may be word processors, spreadsheets, web browsers, or the like and the hardware device may be a printer, memory, Universal Serial Bus (USB) port or any other hardware device. In Linux or Unix operating systems, various functional "layers" are typically provided between application programs and various hardware devices or peripherals. The layer[[s]] <u>that interacts</u> with the hardware and manages applications is the kernel. The shell loads and executes application programs that the kernel manages. The kernel interacts with the hardware devices via a driver associated with each particular hardware device. Thus, not only must the driver be customized for its associated hardware, it must also be customized for the kernel from which it receives instructions and commands. The kernel typically exports Application Program Interface (API) commands to the driver. In Linux, these API commands include identification data which identifies the version of the kernel. On the other hand, the driver exports a version string to the kernel, the version string defining identification data required to establish a version match between the driver and kernel for operation of the driver with the kernel. Linux Device drivers currently available in the market-place rely on the identification data for proper operation and, when the kernel changes, e.g. a newer version is released, the driver requires updating as well. Thus, even though no new functionality has been introduced to the

driver, it still requires recompilation of the device driver source code if the version of the kernel has changed.

Please replace paragraphs **[0014]** – **[0016]** with the following amended paragraphs:

[0014] The kernel 16 defines the heart of the Linux or UNIX operating system and, under its control, the shell 14 interprets user commands of the application programs 12 whereupon the kernel 16 exports various application program interfaces (APIs) to the relevant device driver 18 for execution. For example, if a user executes an exec command from one of the applications 12, the shell 14 interprets the exec command and communicates it to the kernel 16, which in the Linux environment, then converts the command into a exec kernel API which then effects the execution of a process. Typically, each device 20 is viewed as a file system and the device drivers 18 communicate the data in a binary format to the file system interface via specific APIs.[[.]] Accordingly, each device driver 18 is typically configured for the specific hardware that it drives and, since the device driver 18 obtains its commands from the kernel 16, the device driver 18 and the kernel 16 must be configured to function with each other.

[0015] Referring in particular to **Figures 2** and **3** of the drawings, each kernel 16 has a kernel version 23 that identifies the particular version of the kernel 16. The kernel version 23 has unique symbols 24 ~~which~~ that control which version of APIs are exported to device drivers. [[, w]] When compiling a device driver 18, ~~to match a device driver to a specific kernel version~~ a header 26 is included in the compilation of the device driver 18 to match a device driver to a specific kernel version. The symbols 24 are uniquely associated with the kernel version 23 and are used to ensure that the driver 18 only runs on the matching Linux kernel

version 23 for which it has been compiled. The device drivers 18, illustrated in **Figure 2** of the drawings, are dynamic device drivers which have not been compiled into the kernel itself but function as stand-alone drivers that are in an object format or .o format and which are run by the kernel 16 when loaded by the administrator of the computer.

[0016] If the version of the APIs used in a device driver [[do]] does not match the version of the APIs exported by the Linux Kernel, the device driver will not dynamically load. A user is then required to obtain the version of the device driver 18 associated with the kernel version 24. This inability to load typically occurs upon release of a different version of Linux. If the developer has released the source code of the particular driver, the user may then obtain the new version of the kernel and recompile the device driver 18 using the new version. However, if the developer has only released the binary version of the device driver 18, the developer would need to code, compile, and distribute a new device driver for operation with the new version of the kernel. In particular, a new driver would be coded with the appropriate functionality as shown at step 28 in **Figure 3**, whereafter the driver is then compiled by the developer using the kernel symbols 24 to produce a header 26 (see step 30 in **Figure 3**) which corresponds with the particular kernel version 23, thereby generating a new device driver 32 specifically for use with the new version of the kernel. The compiled version of the completed device driver is then distributed.

Please replace paragraph **[0021]** with the following amended paragraph:

[0021] The generic device driver, in accordance with the invention, is configured to operate independent of the kernel version. Referring in particular to **Figures 5** and **6** of the drawings, when a developer wishes to release a new version of a driver, the functionality for the

particular driver is obtained and the various APIs to be included in the driver are coded as shown at step 52. Once the device driver has been coded, the coded driver is then compiled (see step 56) in a different manner to produce a generic device driver component 54 (see **Figure 6**) that does not include a header 26 associated with any particular version of a Linux kernel 16. The driver is thus compiled, as described in more detail below, to generate an incomplete generic, and kernel version independent (KVI), driver component 54 in step 56. The incomplete generic device driver component 54 is typically in the form of an object file or .o file and defines a computer program module for use with a master computer program defined by the Linux operating system. Thereafter, the user runs an installation package 58 on the computer system 22 to generate the customized KVI device driver 50 (driver .o). The method of generating the generic, kernel version independent, device driver component 54 and operation of the installation package 58 is described in more detail below.

Please replace paragraph **[0035]** with the following amended paragraph:

[0035] Make sure the loader passed in the address of module base [[as]] is a parameter to this driver.

```
        if ((modBase == 0) || (modBase==-1))
        {
#ifdef DEBUG
                printk("%s: usage: insmod %s.o modBase=<number> \n",
                        MODULE_NAME, MODULE_NAME );
#endif
                return(EINVAL);
        }
```

Please replace paragraph **[0043]** with the following amended paragraph:

[0043] If ~~anunload~~ an unload is requested, it could be that all the functions were not imported

that were needed to ensure that the local pointers exist before they are used. This step is included as a safety feature.

```
if (g_sDev)
{
```

Please replace paragraphs **[0046]** and **[0047]** with the following amended paragraphs:

[0046] Referring in particular to **Figure 8** of the drawings, the user is provided with the generic device driver component 54 (see step 60) (driver .o) as well as the installation package 58 (see step 62), whereafter the generic device driver component 54 and the installation package 58 are installed on the Linux system (see step 64) of the computer system 22. Thereafter, the user runs a makefile that generates an object file (version .o) (see step 66) associated with the particular version of the Linux kernel 16 on the computer system 22. The makefile is run to link the version.o with the driver.o object files as shown at step 68. During the above-mentioned steps, the installation package 58 links the particular version of the Linux kernel 16 on the computer system 22 with the driver component 54 and runs a make install which gets the kernel specific address (kernel symbols 24) of the module list and passes this to the generic device driver component 54 as shown at step 70 to produce a customized kernel version independent (KVI) device driver 50. The KVI device driver 50 is then loaded on the kernel 16 as shown at step 72, whereafter the device driver binary finds a module list export head as shown at step 74. When the Linux kernel exports commands to the customized device driver 50, the customized device driver 50 imports the APIs that is uses and ignores the kernel version data in the API (see step 76). The customized device driver 50 then runs inside the kernel 16 as shown in step 78.

[0047] The life cycle of a customized device driver 50 is shown in **Figure 7**. The developer

uses the kernel version independent (KVI) method (as described above) to generate source code for the device driver component 54 (see step 80) that, as shown in step 82, is then complied to an object file (see step 82) that defines the generic device driver component 54. The device driver component 54, together with the installation package 58, is then shipped or supplied to the users for installation on the computer system 22. For example, a system administrator may load the device driver component 54 and the installation package 58 on to the computer system 22 (see step 84) as described in more detail above with reference to **Figure 8**.

Please replace paragraph **[0053]** with the following amended paragraph:

[0053] **Figure 10** **[[9]]** shows a diagrammatic representation of a machine in the exemplary form of the computer system 22 within which a set of instructions, for causing the machine to perform any one of the methodologies discussed above, may be executed. In alternative embodiments, the machine may comprise a network router, a network switch, a network bridge, Personal Digital Assistant (PDA), a cellular telephone, a web appliance or any machine capable of executing a sequence of instructions that specify actions to be taken by that machine.

SUBSTITUTE SHEET
(please insert at end of detailed description but before claims)

Computer Listing A:

```
STATIC loff_t sym_lseek( struct file *f, loff_t off, int a)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC ssize_t sym_read( struct file *f, char *c, size_t b, loff_t * a)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV ); }
STATIC ssize_t sym_write(struct file *f, const char *c,size_t b,loff_t * a)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC int sym_readdir(  struct file *f, void *v, filldir_t dir)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV);    }
STATIC unsigned int sym_poll( struct file *f,  poll_table *poll)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV);   }
STATIC int sym_ioctl(struct inode *i, struct file *f, unsigned int cmd,
        unsigned long arg )
{  m_printk("%s: unsupported function %s cmd %d \n",MODULE_NAME,
__FUNCTION__, cmd );
        return(ENODEV); }
STATIC int sym_mmap( struct file *f, struct vm_area_struct *vm)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC int sym_open ( struct inode *i, struct file *f)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC int sym_flush( struct file *f)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC int sym_release (struct inode *i, struct file *f)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC int sym_fsync( struct file *f, struct dentry *d)
```

```
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC int sym_fasync(int b,  struct file *f, int a)
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC int sym_check_media_change( kdev_t dev )
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC int sym_revalidate( kdev_t dev )
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }
STATIC int sym_lock( struct file *f, int a, struct file_lock *l )
{       m_printk("%s: unsupported function %s \n",MODULE_NAME,
__FUNCTION__ );
        return(ENODEV); }

STATIC struct file_operations sym_opts =
{
        sym_lseek,
        sym_read,
        sym_write,
        sym_readdir,
        sym_poll,
        sym_ioctl,
        sym_mmap,
        sym_open,
        sym_flush,
        sym_release,
        sym_fsync,
        sym_fasync,
        sym_check_media_change,
        sym_revalidate,
        sym_lock
};
```